Learning by Porting a Perl 5 module to Perl 6

Jonathan Worthington

The Plan

Take the `CSS::Tiny` CPAN module by ADAMK

Chose a module by a known experienced Perl 5 programmer, in hope of showing good Perl 5 being translated into good Perl 6

Fairly typical mix of text processing, basic OO programming, a little file I/O, and straightforward data structures

CSS::Tiny

Basic CSS parser and generator

Can parse and/or save to a file using `read` and `write` methods

Also can parse a string or produce one, using `read_string` and `write_string`

These create objects, which function like a 2 level hash: styles at the top level, and the properties within each style's hash

`use strict;` is the default

Therefore, we can simply remove it, and have the familiar protections.

```
Tiny.pm
@@ -1,6 +1,5 @@
 package CSS::Tiny;

-use strict;
 BEGIN {
     require 5.004;
     $CSS::Tiny::VERSION = '1.19';
```

It's a `class`, so say so

In Perl 6, we have a `class` keyword for denoting classes.

```
Tiny.pm
@@ -1,4 +1,4 @@
-package CSS::Tiny;
+class CSS::Tiny;

 BEGIN {
     require 5.004;
```

Versions are part of the class name.

We attach them using the `:ver<...>` adverb. Adverbs, using the colon pair syntax, show up in many places in Perl 6. For example, the + infix operator is really named `infix:<+>`.

```
Tiny.pm
@@ -1,8 +1,7 @@
-class CSS::Tiny;
+class CSS::Tiny:ver<1.19>;

 BEGIN {
     require 5.004;
-    $CSS::Tiny::VERSION = '1.19';
     $CSS::Tiny::errstr  = '';
 }
```

We inherit a default `new` constructor

This is inherited from `Mu`, the base of all classes. Therefore, we can simply delete it.

```
Tiny.pm
@@ -6,9 +6,6 @@ BEGIN {
     $CSS::Tiny::errstr  = '';
 }

-# Create an empty object
-sub new { bless {}, shift }
-
 # Create an object from a file
 sub read {
     my $class = shift;
```

Basic method transformation

We replace `sub` with `method`. There's no need to unpack the invocant manually; it is available using the symbol `self`.

Tiny.pm

```
@@ -107,8 +107,8 @@ sub write_string {
 }

 # Generate a HTML fragment for the CSS
-sub html {
-    my $css = $_[0]->write_string or return '';
+method html {
+    my $css = self.write_string or return '';
     return "<style type=\"text/css\">\n<!--\n${css}-->\n</style>";
 }
```

Closure interpolation

In Perl 6, we no longer have the `${...}` syntax. In fact, here we've no great need to use anything other than the variable to interpolate, as there is no ambiguity. However, we can interpolate blocks inside double-quoted strings in Perl 6, and here may do it for readability.

```
Tiny.pm
@@ -109,7 +109,7 @@ sub write_string {
 # Generate a HTML fragment for the CSS
 method html {
     my $css = self.write_string or return '';
-    return "<style type=\"text/css\">\n<!--\n${css}-->\n</style>";
+    return "<style type=\"text/css\">\n<!--\n{$css}-->\n</style>";
 }

 # Generate an xhtml fragment for the CSS
```

Rinse, repeat

We perform the same pair of transforms again on the `xhtml` method.

Tiny.pm

```
@@ -113,9 +113,9 @@ method html {
 }

 # Generate an xhtml fragment for the CSS
-sub xhtml {
-    my $css = $_[0]->write_string or return '';
-    return "<style type=\"text/css\">\n/* <![CDATA[ */\n${css}/* ]]> */\n</style>";
+method xhtml {
+    my $css = self.write_string or return '';
+    return "<style type=\"text/css\">\n/* <![CDATA[ */\n{$css}/* ]]> */\n</style>";
 }

 # Error handling
```

Signatures eliminate unpacking and checks

Once again, we can use `method`, but this time we give it a signature. This saves us a line of validation.

```
Tiny.pm
@@ -7,11 +7,8 @@ BEGIN {
 }

 # Create an object from a file
-sub read {
-    my $class = shift;
-
+method read($file) {
     # Check the file
-    my $file = shift or return $class->_error( 'You did not specify a file name' );
     return $class->_error( "The file '$file' does not exist" )          unless -e $f
ile;
     return $class->_error( "'$file' is a directory, not a file" )       unless -f _;
     return $class->_error( "Insufficient permissions to read '$file'" ) unless -r _;
```

Just `fail` it

We often want to present the caller with a choice of whether to die with an exception or return an undefined value. In Perl 6, `Failure` fills this role, serving as a lazy exception. Testing it as a boolean or for definedness "disarms" it; trying to use it will throw.

```
Tiny.pm
@@ -9,9 +9,9 @@ BEGIN {
 # Create an object from a file
 method read($file) {
     # Check the file
-    return $class->_error( "The file '$file' does not exist" )          unless -e $f
ile;
-    return $class->_error( "'$file' is a directory, not a file" )       unless -f _;
-    return $class->_error( "Insufficient permissions to read '$file'" ) unless -r _;
+    fail "The file '$file' does not exist"          unless -e $file;
+    fail "'$file' is a directory, not a file"       unless -f _;
+    fail "Insufficient permissions to read '$file'" unless -r _;

     # Read the file
     local $/ = undef;
```

Tossing custom error infrastructure

Thanks to our choice to adopt `fail`, we can dispose of some of the Perl 5 lazy error boilerplate.

Tiny.pm

```
@@ -2,10 +2,6 @@ use v6;

 class CSS::Tiny:ver<1.19>;

-BEGIN {
-    $CSS::Tiny::errstr  = '';
-}
-
 # Create an object from a file
 method read($file) {
     # Check the file
@@ -115,8 +111,4 @@ method xhtml {
     return "<style type=\"text/css\">\n/* <![CDATA[ */\n{$css}/* ]]> */\n</style>";
 }

-# Error handling
-sub errstr { $CSS::Tiny::errstr }
-sub _error { $CSS::Tiny::errstr = $_[1]; undef }
-
 1;
```

Being less stringly typed

Strings don't magically pun as filenames everywhere in Perl 6. Instead we should use `.IO` to turn a string path into an IO handle. Then we do the checks on it as methods. The `given` syntax helps here, and we don't have the `-e` special-case syntax, nor the `_` trick, to remember now. Note that `.e` is really `$_.e`.

```
Tiny.pm
@@ -5,9 +5,11 @@ class CSS::Tiny:ver<1.19>;
 # Create an object from a file
 method read($file) {
     # Check the file
-    fail "The file '$file' does not exist"          unless -e $file;
-    fail "'$file' is a directory, not a file"       unless -f _;
-    fail "Insufficient permissions to read '$file'" unless -r _;
+    given $file.IO {
+        fail "The file '$file' does not exist"          unless .e;
+        fail "'$file' is a directory, not a file"       unless .f;
+        fail "Insufficient permissions to read '$file'" unless .r;
+    }

     # Read the file
     local $/ = undef;
```

try to slurp

We can replace all of the file reading code with a simple `slurp` - but that throws exceptions, and the module should either do that or be consistent in its lazy failing. So, we'll wrap it in a `try`, catching exceptions and leaving them in `$!`.

```
Tiny.pm
@@ -12,10 +12,7 @@ method read($file) {
     }

     # Read the file
-    local $/ = undef;
-    open( CSS, $file ) or return $class->_error( "Failed to open file '$file': $!" )
;
-    my $contents = <CSS>;
-    close( CSS );
+    my $contents = try { slurp($file) } orelse fail $!;

     $class->read_string( $contents )
 }
```

Tweak the method call

To finish the porting of `read`, we simply need to port the method call to the new `.` syntax, and call on `self`.

```
Tiny.pm
@@ -14,7 +14,7 @@ method read($file) {
     # Read the file
     my $contents = try { slurp($file) } orelse fail $!;

-    $class->read_string( $contents )
+    self.read_string($contents)
 }

 # Create an object from a string
```

`write` gets the same treatment

Signatures, file writing logic replaced with `spurt`, use `fail`.

```
Tiny.pm
@@ -68,14 +68,8 @@ sub clone {
 END_PERL

 # Save an object to a file
-sub write {
-    my $self = shift;
-    my $file = shift or return $self->_error( 'No file name provided' );
-
-    # Write the file
-    open( CSS, '>'. $file ) or return $self->_error( "Failed to open file '$file' fo
r writing: $!" );
-    print CSS $self->write_string;
-    close( CSS );
+method write($file) {
+    try { spurt($file, self.write_string) } orelse fail $!;
 }

 # Save an object to a string
```

Holy `shift`! What's that do?!

Turns out `read_string` wants to work on both instances and classes. We can re-write this idiom a bit more clearly - and not risk missing out on any construction logic we might add to our class later.

```
Tiny.pm
@@ -18,8 +18,8 @@ method read($file) {
 }

 # Create an object from a string
-sub read_string {
-    my $self = ref $_[0] ? shift : bless {}, shift;
+method read_string {
+    my $self = self // self.new;

     # Flatten whitespace and remove /* comment */ style comments
     my $string = shift;
```

2 become 1

The first thing that happens is duplicate whitespace and comments are stripped. We can write this quite easily as a single pass over the string. It gets less backslashy with Perl 6's regex syntax, too.

```
Tiny.pm
@@ -18,13 +18,11 @@ method read($file) {
 }

 # Create an object from a string
-method read_string {
+method read_string($string) {
     my $self = self // self.new;

     # Flatten whitespace and remove /* comment */ style comments
-    my $string = shift;
-    $string =~ tr/\n\t/  /;
-    $string =~ s!/\*.*?\*\/!!g;
+    $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;

     # Split into styles
     foreach ( grep { /\S/ } split /(?<=\})/, $string ) {
```

is copy

Parameters are passed read-only by default. This is important for a range of optimizations, prevents various mistakes, and leads to less action at a distnace. Here, though, we want our own copy to work on; this allows the substitution to work out.

```
Tiny.pm
@@ -18,7 +18,7 @@ method read($file) {
 }

 # Create an object from a string
-method read_string($string) {
+method read_string($string is copy) {
     my $self = self // self.new;

     # Flatten whitespace and remove /* comment */ style comments
```

`foreach` **becomes** `for`

There is no `foreach` keyword in Perl 6. `for` is always iterating a list; the C-style for loop is now spelled `loop`. Also, we no longer need parentheses around what we'll loop over.

Tiny.pm

```
@@ -25,7 +25,7 @@ method read_string($string is copy) {
     $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;

     # Split into styles
-    foreach ( grep { /\S/ } split /(?<=\})/, $string ) {
+    for grep { /\S/ } split /(?<=\})/, $string {
         unless ( /^\s*([^{]+?)\s*\{(.*)\}\s*$/ ) {
             return $self->_error( "Invalid or unexpected style data '$_'" );
         }
```

Cuter `grep`; optional method call transform

Next, we can eliminate a closure by realizing that `grep` may receive a regex directly in Perl 6. It can also be nice to rewrite this using method syntax, which better conveys the order of operations.

```
Tiny.pm
@@ -25,7 +25,7 @@ method read_string($string is copy) {
     $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;

     # Split into styles
-    for grep { /\S/ } split /(?<=\})/, $string {
+    for $string.split(/(?<=\})/).grep(/\S/) {
         unless ( /^\s*([^{]+?)\s*\{(.*)\}\s*$/ ) {
             return $self->_error( "Invalid or unexpected style data '$_'" );
         }
```

Saner lookaheads

The lookahead syntax has changed. It's a little longer - but a lot easier to figure out what it means!

```
Tiny.pm
@@ -25,7 +25,7 @@ method read_string($string is copy) {
     $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;

     # Split into styles
-    for $string.split(/(?<=\})/).grep(/\S/) {
+    for $string.split(/<?after '}'>/).grep(/\S/) {
         unless ( /^\s*([^{]+?)\s*\{(.*)\}\s*$/ ) {
             return $self->_error( "Invalid or unexpected style data '$_'" );
         }
```

`unless` goes parenless, use `fail`

The lack of need to put parentheses on `for` extends to conditionals also. Then, we do the now-familiar error-reporting transformation.

Tiny.pm

```
@@ -26,8 +26,8 @@ method read_string($string is copy) {

     # Split into styles
     for $string.split(/<?after '}'>/).grep(/\S/) {
-        unless ( /^\s*([^{]+?)\s*\{(.*)\}\s*$/ ) {
-            return $self->_error( "Invalid or unexpected style data '$_'" );
+        unless /^\s*([^{]+?)\s*\{(.*)\}\s*$/ {
+            fail "Invalid or unexpected style data '$_'";
         }

         # Split in such a way as to support grouped styles
```

Regexes, spaced out

In Perl 6, spaces in regexes are just syntax. Put another way, `/x` is the default. So, before we go porting, let's make it easier to read.

```
Tiny.pm
@@ -26,7 +26,7 @@ method read_string($string is copy) {

     # Split into styles
     for $string.split(/<?after '}'>/).grep(/\S/) {
-        unless /^\s*([^{]+?)\s*\{(.*)\}\s*$/ {
+        unless /^ \s* ([^{]+?) \s* \{ (.*) \} \s* $/ {
             fail "Invalid or unexpected style data '$_'";
         }
```

Quoting rather than backslashing

You can put things in single, or double (interpolating) quotes in Perl 6 regexes. While the backslash approach also works, I tend to find the quotes more readable.

```
Tiny.pm
@@ -26,7 +26,7 @@ method read_string($string is copy) {

     # Split into styles
     for $string.split(/<?after '}'>/).grep(/\S/) {
-        unless /^ \s* ([^{]+?) \s* \{ (.*) \} \s* $/ {
+        unless /^ \s* ([^{]+?) \s* '{' (.*) '}' \s* $/ {
             fail "Invalid or unexpected style data '$_'";
         }
```

Character class changes

We stole `[ . . . ]` for non-capturing groups, so character classes are now `< [ . . . ] >`. Negating a character class now goes outside of it, using –, thus giving `<– [ . . . ] >`. (You can actually add and subtract them, set-like, within the `< . . . >`.)

Tiny.pm

```
@@ -26,7 +26,7 @@ method read_string($string is copy) {

     # Split into styles
     for $string.split(/<?after '}'>/).grep(/\S/) {
-        unless /^ \s* ([^{]+?) \s* '{' (.*) '}' \s* $/ {
+        unless /^ \s* (<-[{]>+?) \s* '{' (.*) '}' \s* $/ {
             fail "Invalid or unexpected style data '$_'";
         }
```

More whitespace fiddling

It's curious this is needed, as we tried to deal with multiple bits of whitespace earlier. In fact, parsing this way makes things hard to reason about generally. But for now, we'll just blindly port the line.

```
Tiny.pm
@@ -33,7 +33,7 @@ method read_string($string is copy) {
          # Split in such a way as to support grouped styles
          my $style      = $1;
          my $properties = $2;
-         $style =~ s/\s{2,}/ /g;
+         $style ~~ s:g/\s ** 2..*/ /;
          my @styles = grep { s/\s+/ /g; 1; } grep { /\S/ } split /\s*,\s*/, $style;
          foreach ( @styles ) { $self->{$_} ||= {} }
```

Match variable changes

Positional matches are now numbered from 0, not 1. In fact, `$0` and `$1` are really just `$/[0]` and `$/[1]`, where `$/` is a `Match` object representing the match. `$0` and `$1` are objects too; we just want strings, and apply the ~ stringification prefix.

```
Tiny.pm
@@ -31,8 +31,8 @@ method read_string($string is copy) {
         }

         # Split in such a way as to support grouped styles
-        my $style      = $1;
-        my $properties = $2;
+        my $style      = ~$0;
+        my $properties = ~$1;
         $style ~~ s:g/\s ** 2..*/ /;
         my @styles = grep { s/\s+/ /g; 1; } grep { /\S/ } split /\s*,\s*/, $style;
         foreach ( @styles ) { $self->{$_} ||= {} }
```

Familiar transforms

We dig into the next line by applying some familiar transforms: using the method form to order the operations as they will happen, and then passing the regex directly to grep.

```
Tiny.pm
@@ -34,7 +34,7 @@ method read_string($string is copy) {
          my $style      = ~$0;
          my $properties = ~$1;
          $style ~~ s:g/\s ** 2..*/ /;
-         my @styles = grep { s/\s+/ /g; 1; } grep { /\S/ } split /\s*,\s*/, $style;
+         my @styles = $style.split(/\s*,\s*/).grep(/\S/).grep({ s/\s+/ /g; 1; });
          foreach ( @styles ) { $self->{$_} ||= {} }

          # Split into properties
```

Regex syntax: easier rules

What chars are syntax in regexes, and which are literal? In Perl 6, it's easy: any word characters are literal, and the rest are syntax. This means we need to backslash or quote a comma.

```
Tiny.pm
@@ -34,7 +34,7 @@ method read_string($string is copy) {
          my $style      = ~$0;
          my $properties = ~$1;
          $style ~~ s:g/\s ** 2..*/ /;
-         my @styles = $style.split(/\s*,\s*/).grep(/\S/).grep({ s/\s+/ /g; 1; });
+         my @styles = $style.split(/\s* ',' \s*/).grep(/\S/).grep({ s/\s+/ /g; 1; });
          foreach ( @styles ) { $self->{$_} ||= {} }

          # Split into properties
```

The case of the always-true `grep`

The final use of `grep` is a little curious: it does a side-effect and then always returns true. The reason? `s///` in Perl 5 returns how many replacements were done, not the resulting string! In Perl 6, we give you back the result; check `$/` after to see if anything got replaced.

```
Tiny.pm
@@ -34,7 +34,7 @@ method read_string($string is copy) {
         my $style      = ~$0;
         my $properties = ~$1;
         $style ~~ s:g/\s ** 2..*/ /;
-        my @styles = $style.split(/\s* ',' \s*/).grep(/\S/).grep({ s/\s+/ /g; 1; });
+        my @styles = $style.split(/\s* ',' \s*/).grep(/\S/).map({ s:g/\s+/ / });
         foreach ( @styles ) { $self->{$_} ||= {} }

         # Split into properties
```

Behaving like a hash

Perl 6 objects are not just hashes underneath. They are an opaque data structure, efficiently laid out in memory and - post-optimization - accessed by cheap pointer offsets. But here, we want to behave like a hash. So we declare a hash attribute, and forward calls on us to the various hash-y methods onwards to it.

```
Tiny.pm
@@ -2,6 +2,8 @@ use v6;

 class CSS::Tiny:ver<1.19>;

+has %!styles handles <at_key assign_key list pairs keys values kv>;
+
 # Create an object from a file
 method read($file) {
     # Check the file
```

Implicit dereferencing

Since we have now made hash access related methods delegate to our hash attribute, and since no explicit dereference is needed in Perl 6, the initialization loop can become a little simpler.

```
Tiny.pm
@@ -37,7 +37,7 @@ method read_string($string is copy) {
         my $properties = ~$1;
         $style ~~ s:g/\s ** 2..*/ /;
         my @styles = $style.split(/\s* ',' \s*/).grep(/\S/).map({ s:g/\s+/ / });
-        foreach ( @styles ) { $self->{$_} ||= {} }
+        for @styles { $self{$_} //= {} }

         # Split into properties
         foreach ( grep { /\S/ } split /\;/, $properties ) {
```

`write_string` becomes a method

Once again, the `shift` of `$self` goes away

```
Tiny.pm
@@ -73,8 +73,7 @@ method write($file) {
 }

 # Save an object to a string
-sub write_string {
-    my $self = shift;
+method write_string {

     # Iterate over the styles
     # Note: We use 'reverse' in the sort to avoid a special case related
```

`for` loops with pointy blocks

To name the loop variable, rather than having it in `$_`, we use the pointy block syntax. This is also the way a lambda is written in Perl 6. We also update some `$self` usages to `self`.

Tiny.pm

```
@@ -80,10 +80,10 @@ method write_string {
     # to A:hover even though the file ends up backwards and looks funny.
     # See http://www.w3.org/TR/CSS2/selector.html#dynamic-pseudo-classes
     my $contents = '';
-    foreach my $style ( reverse sort keys %$self ) {
+    for self.keys.sort.reverse -> $style {
         $contents .= "$style {\n";
-        foreach ( sort keys %{ $self->{$style} } ) {
-            $contents .= "\t" . lc($_) . ": $self->{$style}->{$_};\n";
+        for self{$style}.keys.sort {
+            $contents .= "\t" . lc($_) . ": self{$style}{$_};\n";
         }
         $contents .= "}\n";
     }
```

Interpolation changes

Since blocks now interpolate in Perl 6 strings, we need to escape the `{` inside of the string (we could have used an alternative quoting construct too). Also, `self` will not interpolate; here we put block interpolation to good use.

Tiny.pm

```
@@ -81,9 +81,9 @@ method write_string {
     # See http://www.w3.org/TR/CSS2/selector.html#dynamic-pseudo-classes
     my $contents = '';
     for self.keys.sort.reverse -> $style {
-        $contents .= "$style {\n";
+        $contents .= "$style \{\n";
         for self{$style}.keys.sort {
-            $contents .= "\t" . lc($_) . ": self{$style}{$_};\n";
+            $contents .= "\t" . lc($_) . ": {self{$style}{$_}};\n";
         }
         $contents .= "}\n";
     }
```

Concatenation is now ~

Since the . was stolen for method calls, concatenation is now done with the ~ operator.

```
Tiny.pm
@@ -81,11 +81,11 @@ method write_string {
     # See http://www.w3.org/TR/CSS2/selector.html#dynamic-pseudo-classes
     my $contents = '';
     for self.keys.sort.reverse -> $style {
-        $contents .= "$style \{\n";
+        $contents ~= "$style \{\n";
         for self{$style}.keys.sort {
-            $contents .= "\t" . lc($_) . ": {self{$style}{$_}};\n";
+            $contents ~= "\t" ~ lc($_) ~ ": {self{$style}{$_}};\n";
         }
-        $contents .= "}\n";
+        $contents ~= "}\n";
     }

     return $contents;
```

Finally, the `clone` method needs work

First, we simplify: remove the `Clone` module reference and `eval`, and keep the code.

```
Tiny.pm
@@ -51,10 +51,8 @@ method read_string($string is copy) {
     $self
 }

-# Copy an object, using Clone.pm if available
-BEGIN { local $@; eval "use Clone 'clone';"; eval <<'END_PERL' if $@; }
-sub clone {
-    my $self = shift;
+# Copy an object
+method clone {
     my $copy = ref($self)->new;
     foreach my $key ( keys %$self ) {
         my $section = $self->{$key};
@@ -65,7 +63,6 @@ sub clone {
     }
     $copy;
 }
-END_PERL

 # Save an object to a file
 method write($file) {
```

The copy logic can be written more simply

Using .kv to go over the keys and values easily, and then for copying the inner hash, just rely on the hash constructor and flattening to do the work.

```
Tiny.pm
@@ -53,15 +53,11 @@ method read_string($string is copy) {

 # Copy an object
 method clone {
-    my $copy = ref($self)->new;
-    foreach my $key ( keys %$self ) {
-        my $section = $self->{$key};
-        $copy->{$key} = {};
-        foreach ( keys %$section ) {
-            $copy->{$key}->{$_} = $section->{$_};
-        }
-    }
-    $copy;
+    my %styles_copy;
+    for %!styles.kv -> $style, %properties {
+        %styles_copy{$style} = { %properties };
+    }
+    self.new(styles => %styles_copy)
 }

 # Save an object to a file
```

The typical `1;` at module end can go

We simply don't need this in Perl 6. In fact, it even reminds us so by pointing out we have a useless use of the constant 1 in sink (void) context.

```
Tiny.pm
@@ -95,5 +95,3 @@ method xhtml {
     my $css = self.write_string or return '';
     return "<style type=\"text/css\">\n/* <![CDATA[ */\n{$css}/* ]]> */\n</style>";
 }
-
-1;
```

Stub the grammar

A `grammar` is written a lot like a `class`. Inside, we typically put `token`s or `rule`s, which indicate how we parse. The `TOP` rule is the entry point to the grammar.

```
Tiny.pm
@@ -23,6 +23,12 @@ method read($file) {
 method read_string($string is copy) {
     my $self = self // self.new;

+    my grammar SimpleCSS {
+        token TOP {
+            <style>* [ $ || { die "Failed to parse CSS" } ]
+        }
+    }
+
     # Flatten whitespace and remove /* comment */ style comments
     $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;
```

Simply parsing a `style`

We'll capture the name of the style and then its properties. Laying out the `rule` the way a CSS file typically looks is good for readability.

```
Tiny.pm
@@ -27,6 +27,11 @@ method read_string($string is copy) {
          token TOP {
              <style>* [ $ || { die "Failed to parse CSS" } ]
          }
+         token style {
+             \s* (<-[{]>+) '{'
+                 (<-[}]>*)
+             '}' \s*
+         }
      }

      # Flatten whitespace and remove /* comment */ style comments
```

Start moving towards the grammar

We get a tree of match objects from a grammar. We'll go over the list of styles, and get the captures.

```
Tiny.pm
@@ -38,14 +38,10 @@ method read_string($string is copy) {
     $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;

     # Split into styles
-    for $string.split(/<?after '}'>/).grep(/\S/) {
-        unless /^ \s* (<-[{]>+?) \s* '{' (.*) '}' \s* $/ {
-            fail "Invalid or unexpected style data '$_'";
-        }
-
+    for SimpleCSS.parse($string)<style>.list -> $s {
         # Split in such a way as to support grouped styles
-        my $style      = ~$0;
-        my $properties = ~$1;
+        my $style      = ~$s[0];
+        my $properties = ~$s[1];
         $style ~~ s:g/\s ** 2..*/ /;
         my @styles = $style.split(/\s* ',' \s*/).grep(/\S/).map({ s:g/\s+/ / });
         for @styles { $self{$_} //= {} }
```

Parse style names properly

A little more effort than before, but the %% quantifier modifier - which specifies what comes between quantified things - helps a lot.

```
Tiny.pm
@@ -28,10 +28,11 @@ method read_string($string is copy) {
             <style>* [ $ || { die "Failed to parse CSS" } ]
         }
         token style {
-            \s* (<-[{]>+) '{'
+            \s* (<style_name>+ %% [\s* ',' \s* ]) \s* '{'
                 (<-[}]>*)
             '}' \s*
         }
+        token style_name { [ <-[\s,{]>+ ]+ % [\s+] }
     }

     # Flatten whitespace and remove /* comment */ style comments
```

Use the new `style_name` from the grammar

Since the grammar is now extracting style names, we can just use them. Also avoid whitespace re-parsing.

Tiny.pm

```
@@ -41,10 +41,9 @@ method read_string($string is copy) {
     # Split into styles
     for SimpleCSS.parse($string)<style>.list -> $s {
         # Split in such a way as to support grouped styles
-        my $style      = ~$s[0];
+        my $style      = $s[0];
         my $properties = ~$s[1];
-        $style ~~ s:g/\s ** 2..*/ /;
-        my @styles = $style.split(/\s* ',' \s*/).grep(/\S/).map({ s:g/\s+/ / });
+        my @styles     = $style<style_name>.map(~*);
         for @styles { $self{$_} //= {} }

         # Split into properties
```

Parsing properties properly

Now we update the grammar to identify the property keys/values also.

```
Tiny.pm
@@ -29,10 +29,13 @@ method read_string($string is copy) {
          }
          token style {
              \s* (<style_name>+ %% [\s* ',' \s* ]) \s* '{'
-                 (<-[}]>*)
+                 \s* (<property>+ %% [\s* ';' \s* ]) \s*
              '}' \s*
          }
          token style_name { [ <-[\s,{]>+ ]+ % [\s+] }
+         token property {
+             (<[\w.-]>+) \s* ':' \s* (<-[\s;]>+)
+         }
      }

      # Flatten whitespace and remove /* comment */ style comments
```

Clean up the property parsing

Now all the information we need is in the `Match` objects.

```
Tiny.pm
@@ -45,16 +45,13 @@ method read_string($string is copy) {
     for SimpleCSS.parse($string)<style>.list -> $s {
         # Split in such a way as to support grouped styles
         my $style      = $s[0];
-        my $properties = ~$s[1];
+        my $properties = $s[1];
         my @styles     = $style<style_name>.map(~*);
         for @styles { $self{$_} //= {} }

         # Split into properties
-        for $properties.split(';').grep(/\S/) {
-            unless /^ \s* (<[\w._-]>+) \s* ':' \s* (.*?) \s* $/ {
-                fail "Invalid or unexpected property '$_' in style '$style'";
-            }
-            for @styles { $self{$_}{lc $0} = ~$1 }
+        for $properties<property>.list -> $p {
+            for @styles { $self{$_}{lc $p[0]} = ~$p[1] }
         }
     }
```

Refactoring

We no longer really need the positional captures; we can simply use the named ones for styles and properties.

```
Tiny.pm
@@ -28,8 +28,8 @@ method read_string($string is copy) {
             <style>* [ $ || { die "Failed to parse CSS" } ]
         }
         token style {
-            \s* (<style_name>+ %% [\s* ',' \s* ]) \s* '{'
-                \s* (<property>+ %% [\s* ';' \s* ]) \s*
+            \s* <style_name>+ %% [\s* ',' \s* ] \s* '{'
+                \s* <property>+ %% [\s* ';' \s* ] \s*
             '}' \s*
         }
         token style_name { [ <-[\s,{]>+ ]+ % [\s+] }
@@ -44,13 +44,11 @@ method read_string($string is copy) {
     # Split into styles
     for SimpleCSS.parse($string)<style>.list -> $s {
         # Split in such a way as to support grouped styles
-        my $style      = $s[0];
-        my $properties = $s[1];
-        my @styles     = $style<style_name>.map(~*);
+        my @styles = $s<style_name>.map(~*);
         for @styles { $self{$_} //= {} }

         # Split into properties
-        for $properties<property>.list -> $p {
+        for $s<property>.list -> $p {
             for @styles { $self{$_}{lc $p[0]} = ~$p[1] }
```

Prefer names for clarity

We still have two more positional captures. Let's name them, which helps make it clearer what the data is.

```
Tiny.pm
@@ -34,7 +34,7 @@ method read_string($string is copy) {
          }
          token style_name { [ <-[\s,{]>+ ]+ % [\s+] }
          token property {
-             (<[\w.-]>+) \s* ':' \s* (<-[\s;]>+)
+             $<key>=[<[\w.-]>+] \s* ':' \s* $<val>=[<-[\s;]>+]
          }
      }

@@ -49,7 +49,7 @@ method read_string($string is copy) {

          # Split into properties
          for $s<property>.list -> $p {
-             for @styles { $self{$_}{lc $p[0]} = ~$p[1] }
+             for @styles { $self{$_}{lc $p<key>} = ~$p<val> }
          }
      }
```

Whitespace rules!

If we introduce a token named `<ws>` and use `rule` instead of `token`, then `<.ws>` calls are inserted automatically for us where the regex has whitespace.

```
Tiny.pm
@@ -33,9 +33,10 @@ method read_string($string is copy) {
             '}' \s*
         }
         token style_name { [ <-[\s,{]>+ ]+ % [\s+] }
-        token property {
-            $<key>=[<[\w.-]>+] \s* ':' \s* $<val>=[<-[\s;]>+]
+        rule property {
+            $<key>=[<[\w.-]>+] ':' $<val>=[<-[\s;]>+]
         }
+        token ws { \s* }
     }

     # Flatten whitespace and remove /* comment */ style comments
```

Further whitespace cleanups

```
Tiny.pm
@@ -24,13 +24,13 @@ method read_string($string is copy) {
     my $self = self // self.new;

     my grammar SimpleCSS {
-        token TOP {
-            <style>* [ $ || { die "Failed to parse CSS" } ]
+        rule TOP {
+            <?> <style>* [ $ || { die "Failed to parse CSS" } ]
         }
-        token style {
-            \s* <style_name>+ %% [\s* ',' \s* ] \s* '{'
-                \s* <property>+ %% [\s* ';' \s* ] \s*
-            '}' \s*
+        rule style {
+            <style_name>+ %% [ <?> ',' ] '{'
+                <property>+ %% [ <?> ';' ]
+            '}'
         }
         token style_name { [ <-[\s,{]>+ ]+ % [\s+] }
         rule property {
```

Moving comment handling into the grammar

Comments are, really, just a funny kind of whitespace. We can move the comment handling into the grammar also - meaning we now are doing a 1-pass parse of the CSS!

```
Tiny.pm
@@ -36,12 +36,9 @@ method read_string($string is copy) {
         rule property {
             $<key>=[<[\w.-]>+] ':' $<val>=[<-[\s;]>+]
         }
-        token ws { \s* }
+        token ws { \s* | '/*' .+? '*/' }
     }

-    # Flatten whitespace and remove /* comment */ style comments
-    $string ~~ s:g/ \s ** 2..* | '/*' .+? '*/' / /;
-
     # Split into styles
     for SimpleCSS.parse($string)<style>.list -> $s {
         # Split in such a way as to support grouped styles
```

Immutable too!

Earlier we had to add `is copy`, since we were changing `$string` as our first pass. Now we don't do that, so the `is copy` can go away.

Tiny.pm

```
@@ -20,7 +20,7 @@ method read($file) {
 }

 # Create an object from a string
-method read_string($string is copy) {
+method read_string($string) {
     my $self = self // self.new;

     my grammar SimpleCSS {
```

Finally, make the comments match reality

```
Tiny.pm
@@ -39,13 +39,13 @@ method read_string($string) {
         token ws { \s* | '/*' .+? '*/' }
     }

-    # Split into styles
+    # Parse each style.
     for SimpleCSS.parse($string)<style>.list -> $s {
-        # Split in such a way as to support grouped styles
+        # Initialize empty hash per style.
         my @styles = $s<style_name>.map(~*);
         for @styles { $self{$_} //= {} }

-        # Split into properties
+        # Add properties.
         for $s<property>.list -> $p {
             for @styles { $self{$_}{lc $p<key>} = ~$p<val> }
         }
```

Results

Started with 131 lines, ended up with 100.

Initial port - not using grammars - in some ways a fairly mechanical set of steps; after a while, they will become quite natural and - in some cases - maybe even partially automatable.

The move to a grammar needed some more skills, but led to an algorithmic improvement to the code: rather than making multiple passes through the data, we make a single one.

We also better separated concerns better; the parsing was neatly decoupled from the building up of the result hash thanks to grammars and `Match` objects.

v5

One exciting direction for aiding porting is the `v5` module.

Can mix Perl 5 and Perl 6 within a single file, meaning that a bit can be ported at a time

How successful this approach will be depends on the module, and if the port is mostly transliteration or needs a rethink of the module's API

Conclusions

Perl 6 is ready for many kinds of modules to be ported to it

It's also a great way to learn Perl 6

Porting the test suite first can allow working test-first on the port also

Go forth and port!

Thanks for listening!

Have fun!